Program control in R

Base functionality

R.W. Oldford

Control flow

Recall the base data structures in R:

dimensionality	homogeneous contents	heterogeneous contents
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Subsets and individual elements can be selected from these using accessors `[]` (with as many indices are the are dimensions), `[[]]` and `$` for lists. Subsets can be identified by index, name, or as the result of logical expressions (see `?Logical`)

Familiar conditional program control (see `?Control`)

```
# Conditionals
if(cond) expr
if(cond) cons.expr  else  alt.expr
```

can also be used with logical conditions `cond` to evaluate arbitrary expressions `expr`, `cons.expr`, or `alt.expr`. Note multiple expressions must be enclosed in braces `{}` and separated by new lines (or a semi-colon `;`).

When a list of alternatives is available to choose from, then the function `switch(expr, ...)` can be used. It evaluates `expr` and matches it the corresponding element of the list `...` following the expression (see `?switch`).

Control flow - iteration and the `lapply()` family

Standard language constructs `for`, `while`, and `repeat` provide simple iteration:

```
# Looping
for(var in seq) expr
while(cond) expr
repeat expr
break
next
```

Three common ways to construct the sequence `seq` being looped over

- elements of the vector `for (value in values) {}`
- numeric indices of the vector `for (i in seq_along(x)) {}`
- names in a data structure `for (name in names(x)) {}`

Each is just a different case of elements of a vector.

While easy to read (and write) and hence good for maintainable code, `for ()` (and other) loops are inefficient in a scripting language like `R`.

Control flow - iteration and the `lapply()` family

When the number of interations is large, the above looping functions are best avoided.

Instead, **vectorized** alternatives should be used instead. These are usually implemented in `C++` or `C` or some other compilable programming language.

The `lapply()` family of functions (see `?lapply`) is one set of vectorized functions implemented in `C`

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

These apply a function `FUN` to the elements of `X` with `...` supplying any optional arguments of `FUN`.

`lapply` returns a list of the same length as `X`

```
lapply(3:4, function(x) x^2)
```

```
## [[1]]
## [1] 9
##
## [[2]]
## [1] 16
```

Control flow - vector output from `sapply()` and `vapply()`

`sapply()` is a user-friendly version of `lapply` which produces simplified results:

```
sapply(3:4, function(x) x^2)
```

```
## [1]  9 16
```

```
sapply(3:4, function(x) x^2, simplify = FALSE)
```

```
## [[1]]
## [1] 9
##
## [[2]]
## [1] 16
```

and `vapply()` is much like `sapply()` but with a pre-specified type for the return value of the function:

```
vapply(3:4, function(x) paste(c(x, 3*x)^2), FUN.VALUE = character(length = 2))
```

```
##      [,1] [,2]
## [1,] "9"  "16"
## [2,] "81" "144"
```

Note: that `sapply()` tries to be clever and so might return values in a list instead of a vector (e.g. when the `FUN` returns results of different lengths). In contrast, `vapply()` always wants to return a vector and so will generate an error if it cannot (e.g. when a list would work). Consequently `vapply()` may be preferred for programmatic use so that errors are generated; use `sapply()` within a function with care.

Control flow - apply()

apply() is used to apply a function to the margins of an array or a matrix:

```
(x <- array(1:8, dim = c(1, 4, 2)))
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    5    6    7    8
```

```
apply(x, MARGIN = 1, FUN = max)
```

```
## [1] 8
```

```
apply(x, MARGIN = 2, FUN = max)
```

```
## [1] 5 6 7 8
```

```
apply(x, MARGIN = 3, FUN = max)
```

```
## [1] 4 8
```

```
apply(x, MARGIN = c(1,3), FUN = max)
```

```
##      [,1] [,2]
## [1,]    4    8
```

Control flow - sweep()

Often we want to remove a summary statistic from the elements of an array.

`sweep()` is used to apply statistical summary function to the specified margins of an array `x` and then to remove (or sweep) the resulting summary statistics out of the array.

```
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
```

Here `x` is the array to be swept, `STATS` the summary statistics to be swept out, `MARGIN` the vector of indices identifying that part of `x` being summarized, and `FUN` is the method by which the summary statistics are swept from `x`.

For example, we could subtract the median values from every column of the `data.matrix` of the data frame `quakes`:

```
x <- data.matrix(quakes)
summaryStat <- apply(quakes, 2, median)
head(sweep(quakes, MARGIN = 2, STATS = summaryStat, FUN = "-"))
```

```
##      lat  long depth  mag stations
## 1 -0.12  0.21   315  0.2       14
## 2 -0.32 -0.38   403 -0.4      -12
## 3 -5.70  2.69  -205  0.8       16
## 4  2.33  0.25   379 -0.5       -8
## 5 -0.12  0.55   402 -0.6      -16
## 6  0.62  2.90   -52 -0.6      -15
```

Note: For other applications, a different `FUN` (or "broom") might be used to sweep out the value.

Control flow - tapply()

`tapply()` applies a function `FUN` to values of an atomic vector `X` given by each combination of factors supplied as `INDEX`. The array is "ragged" in the sense that not all combinations need have a corresponding subset of values in `X`.

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

For example, with the `SAheart` data from `ElemStatLearn`

```
library(ElemStatLearn)
with(SAheart,
     tapply(sbp, INDEX = list(chd, famhist), FUN = mean))
```

```
##     Absent  Present
## 0 134.9806 136.4896
## 1 142.8594 144.3229
```

```
with(SAheart,
     tapply(sbp, INDEX = list(age = cut(age, 3), tobacco = cut(tobacco, 3)), FUN = mean))
```

```
##               tobacco
## age            (-0.0312,10.4] (10.4,20.8] (20.8,31.2]
##   (15,31.3]          128.3846          NA          NA
##   (31.3,47.7]        134.8561    142.3333          NA
##   (47.7,64]          145.6821    145.9355         168
```

Note that there is not a lot of tobacco use (in cumulative kg) amongst younger people. If there are no observations in `X` for a group defined by the factors of `INDEX`, then `NA` is returned for that group.

Control flow - by()

```
by(data, INDICES, FUN, ..., simplify = TRUE)
```

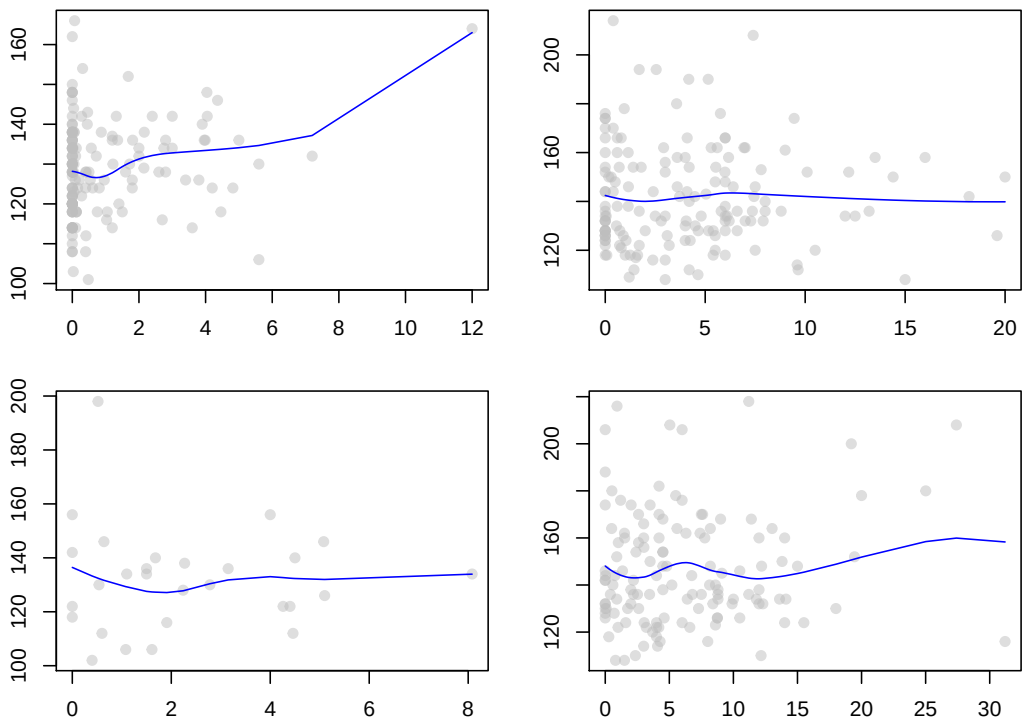
by() is a convenience wrapper function for tapply() which applies the function FUN to values of a data frame data given by each combination of factors supplied as INDICES. Here ... are further arguments to FUN.

```
by(SAheart[,c("age", "tobacco", "obesity")], INDICES = list(chd = SAheart$chd), FUN = summary)
```

```
## chd: 0
##       age           tobacco          obesity     
##  Min.   :15.00   Min.   : 0.000   Min.   :17.75  
##  1st Qu.:27.00   1st Qu.: 0.000   1st Qu.:22.60  
##  Median :40.00   Median : 1.035   Median :25.57  
##  Mean   :38.85   Mean   : 2.635   Mean   :25.74  
##  3rd Qu.:50.75   3rd Qu.: 4.200   3rd Qu.:28.07  
##  Max.   :64.00   Max.   :20.000   Max.   :46.58  
## ------------------------------------------------------------
## chd: 1
##       age           tobacco          obesity     
##  Min.   :17.00   Min.   : 0.000   Min.   :14.70  
##  1st Qu.:42.75   1st Qu.: 1.500   1st Qu.:23.64  
##  Median :53.00   Median : 4.130   Median :26.48  
##  Mean   :50.29   Mean   : 5.525   Mean   :26.62  
##  3rd Qu.:59.00   3rd Qu.: 8.200   3rd Qu.:28.78  
##  Max.   :64.00   Max.   :31.200   Max.   :45.72
```

Control flow - by()

```
savePar <- par(mfrow = c(2,2), mar = rep(2,4))
output <- by(SAheart[, c("tobacco", "sbp")],
             INDICES = list(age = cut(SAheart$age, 2), chd = SAheart$chd),
             FUN = function(data) {
               x <- data[,1]
               y <- data[,2]
               ordered_x <- sort(x)
               fit <- loess( y ~ x, data.frame(x = x, y = y))
               pred <- predict(fit, newdata = data.frame(x = ordered_x))
               plot(data, pch = 19, col = adjustcolor("grey", 0.5))
               lines(ordered_x, pred, col = "blue")
             })
```




```
par(savePar)
```

Control flow - aggregate() for data.frames

```
aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)
```

aggregate() splits the data into groups, computes summary statistics for each, and returns the result in a convenient form.

```
aggregate(SAheart[,c("age", "tobacco", "obesity")],
          by = list(chd = SAheart$chd), FUN = summary)
```

```
##   chd age.Min. age.1st Qu. age.Median age.Mean age.3rd Qu. age.Max.
## 1   0 15.00000    27.00000   40.00000 38.85430    50.75000 64.00000
## 2   1 17.00000    42.75000   53.00000 50.29375    59.00000 64.00000
##   tobacco.Min. tobacco.1st Qu. tobacco.Median tobacco.Mean tobacco.3rd Qu.
## 1     0.000000        0.000000       1.035000     2.634735        4.200000
## 2     0.000000        1.500000       4.130000     5.524875        8.200000
##   tobacco.Max. obesity.Min. obesity.1st Qu. obesity.Median obesity.Mean
## 1    20.000000     17.75000        22.60250       25.57000     25.73745
## 2    31.200000     14.70000        23.63500       26.47500     26.62294
##   obesity.3rd Qu. obesity.Max.
## 1        28.06500     46.58000
## 2        28.78000     45.72000
```

Control flow - mapply()

`mapply()` is a multi-variable version of `sapply()`:

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

where `FUN` is a function to be applied to the elements of some number of arguments, which appear in `...`. The argument `MoreArgs` is a list of other arguments to `FUN` to be used at each call.

```
mapply(rep, 1:3, 3:1)
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 2 2
##
## [[3]]
## [1] 3
```

```
mapply(rep, times = 1:3, x = 3:1)  # using named arguments
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 2 2
##
## [[3]]
## [1] 1 1 1
```

Control flow - functional programming constructs

A number of programming constructs that are common in functional programming are also available (see `?Map`). These are found in many such languages (e.g. dating back to at least `Smalltalk` and `Lisp`). In `R`, these are:

```
Map(f, ...)
Reduce(f, x, init, right = FALSE, accumulate = FALSE)
Filter(f, x)
Find(f, x, right = FALSE, nomatch = NULL)
Position(f, x, right = FALSE, nomatch = NA_integer_)
Negate(f)
```

Here `f` is a function (of the appropriate arity - binary for `Reduce`, unary for `Filter`, `Find`, and `Position`, and k-ary for `Map`), `x` a vector, and `...` any number of vectors.

`Negate(f)` returns a function which when applied to its arguments simply logical negates wheatever would be returned by the (predicate) function `f` (i.e. `Negate(f)` simply returns the function `!f`)

Note: Various functions (e.g. `clusterMap()` and `mcmapply()`) from the package `parallel` provide parallel versions of `Map()`.

Control flow - Map() on a data.frame

```
Map(summary, quakes)
```

```
## $lat
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -38.59  -23.47  -20.30  -20.64  -17.64  -10.72
##
## $long
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   165.7   179.6   181.4   179.5   183.2   188.1
##
## $depth
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    40.0    99.0   247.0   311.4   543.0   680.0
##
## $mag
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    4.00    4.30    4.60    4.62    4.90    6.40
##
## $stations
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.00   18.00   27.00   33.42   42.00  132.00
```

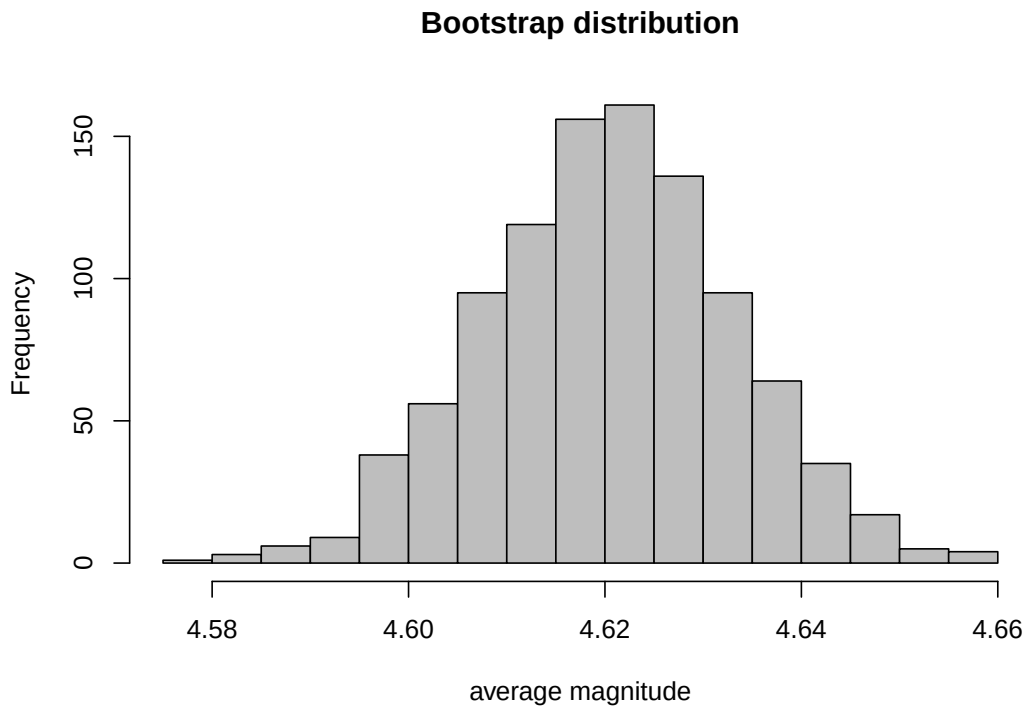
Control flow - `Map()` on samples

Suppose `x` is a vector (i.e. not a list nor a data frame):

```
Bootstrap <- function(x, fun, nreps) {
  n <-length(x)
  Map(function(i) {
    fun(x[sample(seq_along(x), size = n, replace = TRUE)])},
    1:nreps)
}
BootVals <- Bootstrap(quakes$mag, mean, 1000)
```

`Map()` returns a `list` (no simplification) which makes `BootVals` a bit inconvenient for further data analysis:

```
hist(unlist(BootVals), col = "grey", breaks = 15,
     main = "Bootstrap distribution", xlab = "average magnitude" )
```



Control flow - Reduce()

```
Reduce(f, x, init, right = FALSE, accumulate = FALSE)
```

`f` is a function of some number of arguments, `x` is a vector, `init` is an initial value of the same kind as the elements of `x`, `accumulate` is a logical indicating whether results of every reduction step is to be returned or just the last (default), and `right` is a logical indicating whether the reduction should begin at the end (right) of `x` and move backwards:

A simple example might be the calculation of a sample variance, which requires evaluating:

$$\sum_{i=1}^{n} (x_i - \overline{x})^2$$

which requires two passes through `x`, one to get $\overline{x} = \sum x_i/n$, the other to gather the squared differences:

```
twoPass <- function (x) {
  n <- length(x)
  xbar <- Reduce(function(x0, x1) {x0 + x1}, x, init=0) / n
  result <- Reduce(function(x0, x1) {x0 + (x1 - xbar)^2}, x, init=0)
  result
}
n <- 1000 ; x <- rnorm(n)
twoPass(x) / (n-1)
```

```
## [1] 0.9849108
```

Control flow - `Reduce()`

It is not uncommon to see authors suggest a single pass algorithm based on the mathematically equivalent form:

$$\sum_{i=1}^{n}(x_i-\overline{x})^2=\sum_{i=1}^{n}x_i^2-n\overline{x}^2=\sum_{i=1}^{n}x_i^2-\frac{1}{n}\left(\sum_{i=1}^{n}x_i\right)^2$$

which is implementable using a single `Reduce()` when we gather two results as we go:

```
onePass <- function (x) {
  n <- length(x)
  result <- Reduce(function(x0, x1) {x0 + c(x1^2, x1)}, x, init=c(0,0))
  result[1] - (result[2]^2 / n)
}
onePass(x)/(n-1)
```

```
## [1] 0.9849108
```

Unfortunately, the one pass algorithm is not numerically sound. It can even produce negative values! The following corrected two-pass method based on both (replace x_i by $x_i-\overline{x}$ in the one-pass) is sounder numerically than either.

```
twoPassCorrected <- function (x) {
  n <- length(x)
  xbar <- Reduce(function(x0, x1) {x0 + x1}, x, init=0) / n
  result <- Reduce(function(x0, x1) {x0 + c((x1 - xbar)^2, (x1-xbar))}, x, init=c(0,0))
  result[1] - (result[2]^2 / n)
}
twoPassCorrected(x)/(n-1)
```

```
## [1] 0.9849108
```

Control flow - `Filter()`

```
Filter(f, x)
```

`Filter()` selects that subset of the elements of `x` which return `TRUE` to the predicate function `f`.

Note: It is NOT to be confused with the base `R` function `filter()` which applies "linear filtering" to a time series.

Suppose for example, we want only those variates in a dataset which either are "factors" or have at most 3 different values and hence are potential factors.

```
data(SAheart, package = "ElemStatLearn")
possibleFactors <- Filter(f = function (var){
           is.factor(var) | (!is.factor(var) & length(unique(var)) <= 3)},
           SAheart)
possibleFactors[1:3,]
```

```
##   famhist chd
## 1 Present   1
## 2  Absent   1
## 3 Present   0
```

Control flow - Find()

```
Find(f, x, right = FALSE, nomatch = NULL)
```

Find() searches and finds the first element in x returns TRUE to the predicate function f.

```
possibleFactor <- Find(f = function (var){
           !is.factor(var) & length(unique(var)) <= 3},
           SAheart)
str(possibleFactor)
```

```
##  int [1:462] 1 1 0 1 1 0 0 1 0 1 ...
```

Find() returns NULL if no such element is found.

```
Find(f = function (var){is.factor(var) & (nlevels(var) >=5)},
     SAheart)
```

```
## NULL
```

Control flow - Position()

```
Position(f, x, right = FALSE, nomatch = NA_integer_)
```

Position() searches and finds the first element in x returns TRUE to the predicate function f.

```
Position(f = function (var){
  is.factor(var) | (!is.factor(var) & length(unique(var)) <= 3)},
  SAheart)
```

```
## [1] 5
```

Position() returns the value of nomatch if no such element is found.

```
Position(f = function (var){is.factor(var) & (nlevels(var) >=5)},
     SAheart)
```

```
## [1] NA
```